Computer Science Department

TECHNICAL REPORT

ALGORITHM DERIVATION BY TRANSFORMATIONS *

by

Micha Sharir

October 1979

REPORT NO. 021

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

ALGORITHM DERIVATION BY TRANSFORMATIONS *

by

Micha Sharir

October 1979

REPORT NO. 021

* This work is based upon work supported partly by the National Science Foundation under Grant No. MCS-78-18922. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

This work is also based upon work supported in part by the U.S. Department of Energy Contract No. EY-76-C-02-3077.

ABSTRACT

Various issues in the design of a transformational programming system are discussed; in particular we study the issue of passage from a nonprocedural problem specification to a first executable solution of that problem. Then scenarios describing the possible construction of two nontrivial problems - topological sorting and the eight queens problem - are given. Transformations shown to be of particular value are: formal differentiation, backtracking and recursion optimization, and elimination of nondeterminism.

1. INTRODUCTION

The major goal of programming methodology has always been to make the programming process as systematic as possible, thereby producing a framework within which programs can easily be written, debugged, maintained, understood and proved correct. To this end various tools and techniques have been suggested, such as structured programming, high-level languages, abstract data-types, advanced optimization techniques, sophisticated programming environment, etc.

Although these methods have proved very useful in speeding-up the program development process, they still fall short of understanding the essence of programming, which is still viewed more as an art than as a science. We still lack formalization, let alone mechanization, of the process by which problem specifications are turned into efficient and correct programs.

While there are obviously many ingenious steps taken by a programmer (or more often by an algorithm designer) in order to solve a particular problem in an efficient manner (or to solve it at all), it is nevertheless quite obvious that most of the steps involved in the programming process are standard and rather trivial. In fact, several such steps can already be performed by automatic optimization techniques, to which new and more advanced methods are being added continuously. Still, these techniques cover only a small portion of the process of program construction, and many standard programming techniques are still way out of the reach of an automatic or semi-automatic programming system.

There are at least three major motivations for seeking better understanding of the programming process. In order of their immediate applicability, they are:

(a) The ability to describe and express complex algorithms will be greatly enhanced if one is able to outline the way by which these algorithms have been arrived at, rather than just describe the final polished product. For example, the tricky Deutsch-Schorr-Waite marking algorithm (cf. [Kn, p. 417]) is rather difficult to comprehend as a stand-alone algorithm (especially when given in Knuth's relatively low-level style). However if we describe this algorithm as an optimized version of a depth-first search of the given graph, in which the recursive stacking mechanism has been made explicit and optimized by using available pointer space within each node in the list being processed, we can immediately gain much greater insight into the problem. The correctness of the algorithm becomes quite obvious, and we can easily generalize it to handle cases where each node can point to any fixed number of other nodes. Using such a method to describe algorithms also allows us to build 'genealogy trees' for various families of algorithms, to find similarities and differences between various algorithms having a common goal, and sometimes even to discover new specific algorithms from more general ones. Treatises of this sort can be found e.g. in [LGdR], [Sc1] and [DS].

(b) The ability to prove program correctness will also be greatly enhanced if one can formally trace the process of converting a given high-level specification to a low-level program, rather than just be given the final program. Most of the present literature on program verification investigates the problem of verifying a low level program without any information on the way in which it has been constructed. As argued in e.g. [Sc1], this approach is defficient because, among other reasons, it forces the programmer/ verifier to mentally 'decompile' the final program in order to construct the required assertions which are needed for the correctness proof. A more natural approach would be one in which one begins with high-level problem specification which is correct a priori (with respect to itself), and then applies to it a sequence of correctness-preserving transformations, which takes this specification through successive refinements into a low-level detailed version. The correctness of that version is then obvious. This does not suggest that correctness proofs will always be simpler if that second approach is used, but only that they will be more natural and systematic. One might compare this situation to solving a problem in algebra, where one is required to solve an equation such as, say, X**2 = 5776. To prove that 76 is a solution will be much simpler, though less illuminating, than to trace a correct square-root evaluation procedure applied to 5776. In many cases, though, correctness can be guaranteed with little or no proof at all if the 'top-down' construction scheme noted above is followed.

(c) Eventually, when most of the elements involved in program development become better understood, formalized and to some degree automated, it might be possible to build up a semi-automatic programming system, capable of constructing complex programs from their specification, with some aid from the user, in a relatively simple manner that might even prove to be faster than what it would take to write, test and verify such a program directly. Moreover, using such a system will yield the advantages (a) and (b) mentioned above. This somewhat futuristic vision of 'automatic programming' is indeed the ultimate goal being pursued by programming methodologists.

One of the possible approaches to program construction is, as already hinted above, program transformation. In this approach, the programming task is initially given in a relatively high level style, either as a static specification or as a dynamic program. Such high level allows succinct expression of the problem/solution and its correctness is easy to verify (in most cases no verification will be required, especially when the initial program version actually coincides with its specification and so is correct a priori). Next, in order to improve its execution efficiency, this initial version is subject to a sequence of correctness-preserving transformations which can improve the control flow of the program, introduce auxiliary variables for better storage management, remove or optimize recursion and nondeterminism, specify concrete data-structure representations and finally apply standard 'clean-up' optimizations such as common subexpression elimination, constant propagation, code motion, dead code removal etc.

Various classes of program transformations have been studied so far. They include: the folding/unfolding technique of Burstall and Darlington for recursive programs [BD]; various recursion removal schemes [WS]; refinement via abstract data-type definitions [Ba];

automatic selection of data structures [SSS], [Lo] and formal differentiation of set-theoretic expressions [Pa]. This last technique deserves more comment as it turns out to play a central role in the construction of the algorihms that we shall consider in this paper. This technique applies in cases where a complicated and expensive set-theoretic expression is repeatedly computed within a program loop in which the arguments of that expression change only slightly. It is then often possible to replace these repeated computations by much cheaper incremental computations which can be used to update the value of the original expression. This technique turns out to be an extremely powerful algorithm transformation which can improve algorithm efficiency by orders of magnitude, as shown in [Pa] and as will be demonstrated below. Various aspects and applications of this technique are studied in a companion paper [Sh]. Use of program transformations in program verification can be found in [Sc1] and is also being studied in [Dea].

In this paper we will study various issues involved in transformational construction of programs from their specifications. Section 2 will discuss methods to accomplish the first step in such a process - namely the convertion of a static specification to an initial executable version. We will focus our attention on specifications involving set-theoretic objects and investigate in some detail a standard construction method for such objects, namely to construct them incrementally by adding one (or few) elements at a time. Sections 3 and 4 consist of case studies of two nontrivial problems. Section 3 considers the problem of testing a given graph for the existence of cycles and the relation of this problem to Knuth's topological sorting technique [Kn, p. 258]. Section 4 deals with the eight queens problem. In both sections an attempt is being made to systematize the derivation process as much as possible, and to note common techniques (most of which are still rather heuristic) which are likely to have a broad range of applicability.

2. ON IMPLEMENTATION OF NONPROCEDURAL OR 'ONE-STEP' SPECIFICATIONS

As one of the initial steps toward the design of a transformational programming system, this section will consider some of the issues involved in 'implementation' of nonprocedural, or 'one-step' algorithm specification.

We envisage a system able to accept the 'base-form' of algorithms in the form of a very high-level specification. In most cases such specifications will ignore control-flow details, and consist solely of input-output relationships, i.e. will state assumptions concerning the input to the algorithm, and will then specify the required properties of the output object(s) in some form of predicate logic. Typical examples (written in a very tentative specification language; see below for details) are:

(1) sort:

```
      assume A : tuple(integer) 1 ... N;
                 N : integer;
                 N > 0;

      find P : permutation(N) st
          (forall I in {1 ... N-1} : A(P(I)) <= A(P(I+1)) )
```

(2) Find all prime numbers less than some given number:

```
      assume N : integer; N > 0;
      find S : subset {1 ... N} st
          (forall X in {1 ... N} : X in S iff
              (forall Y in { 2 ... X-1} : not divides(Y, X)))
```

(3) Find the transitive closure of a set under a relation:

```
      assume E : set;  R : map(elmt E) elmt E;  S0 : subset E;
      find S : subset E st
          S0 subset S and (forall X in S : R{X} subset S) and
          min(S, inclusion);
```

(4) String pattern matching:

```
      assume  T : string;  P : string;
      find I : integer st
          (forall J in [1 ... # P] : P(J) = T(I+J))
```

As can be noted from such examples, these specifications have the following general structure: Input assumptions tend to resemole data-type declarations, and also include certain relationships between input objects. Output requirements ask for computation of a certain object which must satisfy a rather involved predicate, usually involving quantifications over sets or tuples.

Let us assume for the time being that our initial specification has such a form. Various issues then arise. For example, what (if any) notion of correctness should such a specification possess? This is a nontrivial problem especially because some programming effort has already been involved in formulating the programming task (initially set before the programmer in a different, less formal form) as such a formal statement. Temporarily we will ignore this issue, and assume that the specification itself is the original version of the programming task, and is therefore 'correct' a priori. Any version obtained from this specification by applying a sequence of correctness-preserving transformations will therefore also be correct. (Nevertheless it will be very useful for such specifications to be executable as they stand, so as to strengthen the belief of the programmer in the 'correctness' of this base form of the algorithm. See [DS] for related comments.)

However the main issue concerning such specifications, is how to convert them into procedural form. I.e., given such a specification, we would like to construct from it, as mechanically as possible, a more efficient procedural program that accomplishes the task implied by the specification, which could then be further improved by successive transformations. While we certainly cannot fulfill this goal in all cases, we can hope that systematic study of commonly occuring patterns will enable us to generate 'default' programs automatically for a reasonably wide class of specifications.

Assume that the specification to be considered is given in a form

```
assume G(A1, A2 ... An);
find X st P(X, A1 ... An);
```

where A1 ... An are the input objects of the problem, where G is a predicate describing the input assumptions, where X is the required output object, and where P is a predicate describing the properties that X should satisfy. In what follows we will usually not state the 'assume' part of the specification explicitly, but imagine it to be given implicitly.

Two syntactic extensions to the 'find' part of such specifications suggest themselves. First, let us assume that X is qualified (by conditions constituting part of P) as having a certain (parametrized) 'data-type', whose purpose is to indicate a default search space for X, which should be either finite or at worst enumerable. Such a data-type should be either absolute (involving no parameters), or else parametrized in terms of input objects only. We propose to denote that part of P in a form resembling conventional data-type declarations, and thus propose to write our specification in the form

```
find X : type st P(X);
```

Typical type declarations are:

```
integer
elmt A
subset E
map (elmt E) elmt F
permutation (N)
tuple (elmt E) 1 ... N
```

Built-in knowledge of such types (and also of more general abstract data types) would be a very important aspect of a system like that which we envisage. These additional types might include trees, order relations, one-one maps, partitions, permutations, polynomials and other typical abstract types. This library of data types should of course be extensible (as most of the features of our proposed system should be), so as to allow the user to add his own favorite data types to the system.

The next extension that we consider is appropriate for problems in which one does not just seek any X satisfying a certain property P, but instead wants the smallest (largest, shortest, minimal etc.) X satisfying P. In such cases it is very useful to separate the part of the predicate which specifies the 'extremum condition' involved from the rest of the predicate, in order to improve the succinctness of the specification. To do so, let us assume that the search space for X is the domain of some partial order relation 'ord', which, for simplicity, we assume to belong to some fixed class of frequently used order relations, including e.g.

(a) integers in their usual order,
(b) strings in lexicographical order,
(c) subsets of some given set in inclusion order,
(d) tuples or maps whose range is some partially ordered set in pointwise order,
(e) tuples whose range is partially ordered in lexicographical order.

Then we can use the notation

```
      find X : type st P(X) and EXT(X)
```

where EXT(X) is an 'extremum condition' that X must satisfy. This condition might typically have a form

```
      max (F(X), ord)
```

to indicate that we want an X for which the expression F(X) is maximal in the order 'ord'. Of course instead of 'max' we could also use 'min', 'largest', 'smallest' etc. with obvious meanings. This notation is suggested here only for syntactic convenience and has little impact on the transformations studied in this paper. See however [Sh] for formal transformation rules which can realize such extremum requirements in some important special cases.

Assume that we are given a specification in such a form. Its realization (i.e. construction of a procedural program to compute the required object) will depend on the data type 'type' of X and on the predicate P(X). To understand the force of this remark, we first consider a few simple cases where P(X) has a structure which can be translated into procedural form easily.

Assume first that P(X) has the form X R A, where A is some constant expression (that is, depends only on input objects) and R is a relation.

(a) if R is an equality, then this task can be easily realized by 'X := A;'.

(b) If R is set membership, realize the task by 'X := arb* A;', where arb* denotes nondeterministic selection from A.

(c) In general, X R A may be realized by 'X := arb* {Y : Y R A}'. (Here one should generally use the data-type declaration for X also, to qualify the set appearing in the above selection.)

Assume next that P(X) has a form in which X is not isolated from the input objects in an obvious manner. E.g. let P(X) be 'X = T(X)'. In general this case is substantially harder than the cases considered above, and often there will be little that can be done automatically. There are however several possible courses of action which will be helpful in special cases:

(c.1) The system could prompt the user to re-express P(X) in a form in which X is isolated. This isolation can be accomplished either in a fully manual manner, or else by suggesting some kind of simplification to the system.

(c.2) If the search aims to attain some extremum condition, it might be realized using successive approximation techniques. Fixed point problems in well-founded sets can often be solved in this manner.

Next assume P(X) to be a conjunction of the form 'Q(X) and R(X)'. In this case there are several possibilities:

(e.1) Q(X) and R(X) may be independent, i.e. to realize the conjunction it is sufficient to realize each conjunct independently by procedures that do not conflict with each other, and then combine their outputs in a fixed predetermined manner. For example, to realize

```
      find X : set  st A in X and B in X;
```

one can simply satisfy 'X : set' by assigning any set to X, then to realize 'A in X' by 'X with:= A', and similarly realize 'B in X' by 'X with:= B'.

Likewise, to realize

```
      find X : tuple  st X(1) = A and X(2) = B
```

we can assign any tuple to X, and then set X(1) := A; X(2) := B;

(e.2) It may be possible to simplify 'Q(X) and R(X)' into a form in which the conjunction does not appear explicitly. This will usually make P(X) easier to handle. For example, we can simplify

```
      find X st X in A and X in B
```

into

```
      find X st X in A * B.
```

Similarly, we can simplify

```
      find X st X > A and X > B
```

into

```
      find X st X > max(A, B)
```

etc.
(e.3) If none of the above rules apply, we can use a general strategy which will choose one of the conjuncts, say Q(X), as an 'operational' predicate, i.e. will try actively to realize it, and use the other conjunct R(X) as a test to restrict the possible realizations of Q(X). Heuristically we can say that we want to 'realize Q(X) provided that R(X)'. For example, the task

```
       find X st X in E and F(X) > 0
```

is solved by

```
       X := arb* { W in E st F(W) > 0}
```

A similar treatment can be used to handle disjunction of subpredicates.

These few examples concerning realization of given specifications have been given here only as a prologue to the main issue to be discussed in this section, namely - incremental construction of composite objects from their given specification. In spite of this, a more comprehensive study of ways to realize general specifications is certainly called for.

This last approach (e.3) to conjunction has immediate application in the realization of tasks having the general form

```
       find X : type  st P(X);
```

So far we have ignored the data-type part of such a specification, but in general it must be treated as an additional conjunct of the specification predicate. In most cases, this will be chosen as the operational conjunct in case (e.3), and will therefore be realized in a most general way, using P(X) as a restrictive condition on the object being generated. Since the data types likely to be specified will often belong to a relatively small family of commonly used types, we can appropriately study general schemes to realize objects having such data types in a more systematic manner than other predicates. Such a study will enable our system to obtain an initial procedural version of a given specification, which should then be subject to a process in which a sequence of correctness preserving transformations will be applied to that version till satisfactory efficiency is obtained.

We note that in many programming tasks the object(s) being sought are composite. A program whose task is to compute an integer is either relatively trivial, or else involves a rather highly inventive arguments. A more tractable programming task would be to construct a certain set, list, table etc. of objects, having certain properties.

A key method for construction of a composite object is to build it incrementally, by starting with some initial (usually empty) value, and then by adding elements one at a time, while retaining the validity of the predicate P(X) defining X (or perhaps while aiming to make P(X) true). We will refer to such a method as 'growth of domains of validity'. In such cases the elements of X will often be selected nondeterministically from some domain and the validity of P(X) will be tested each time X is augmented. This schema leads itself to application of formal differentiation of the predicate P(X). Formal differentiation enables us to replace repeated evaluations of P(X) (likely to be very costly) by evaluations of 'incremental' or 'derivative' predicates which will often be much more efficient to evaluate.

We thus want to collect recipes which realize a 'partial' specification of the form

```
(*)    find X : type st  < condition >
```

where 'type' denotes a composite data-type, where the <condition> is left unspecified, and where we aim at an incremental construction of X. We begin with the following typical cases:

```
(1)    find X : subset E
```

This can be realized as

```
        X := {};
        (while arb* {true, false})
            U := arb* (E - X);
            X with:= U;
        end while;
```

REMARK 1: This program is in fact more general than its specification, in the sense that it allows elements to be added to X in any possible order, which will create some redundancy among the resulting subsets. In fact the number of possible ways of executing this program is roughly e * (#E)!, compared with the 2**(#E) possible subsets. Nevertheless it is important to allow this redundancy, because often final algorithm efficiency may greatly depend on the order in which elements are added to X. We want to allow considerable redundancy initially, and to prune the search space later.

REMARK 2: Note that in the preceding schema the predicate 'arb* {true, false}' plays two roles. To see this, assume that the qualifying predicate P(X) is given, so that we want to realize

```
(*)    find X : subset E st P(X)
```

Then exactly the same program could be used, except that the while header should be replaced by

```
        (while not P(X))
```

With this change our code will not find the most general subset of E satisfying P (in fact it will yield only some sequentially minimal such subset). This is quite all right, for the specification as stated did not require us to compute all such subsets. However, if we want to regard (*) as a partial specification to which additional constraints might be added later on, then we should use the following while header:

```
       (while not P(X) or arb* {true, false})
```

This would correspond to the following specification

```
       find X : subset E st P(X) and ?
```

where ? denotes an undefined predicate.

This indicates that the predicate 'arb* {true, false}' can serve both as a default value for a yet unsupplied predicate, and as a syntactic marker in a specification pattern denoting such an unknown predicate. In what follows we will use the notation ? for both purposes.

We now pass to the examination of a second important pattern, namely

```
(2)    find X : tuple(elmt E)  st ?
```

This can be implemented using the scheme

```
       X := [];
       (while not ?)
           U := arb* E;
           X with:= U;
       end while;
```

Here there is no redundancy since the specification requires that a linear augmentation be produced.

Yet another specification of interest is

```
(3)    find X : map(elmt E) elmt F  st ?
```

If we allow maps to be only partially defined, then construction of such X can be viewed as a generalization of the subset case, and so can be realized by the following scheme:

```
       X := {};
       (while not ?)
           U := arb* (E - domain X);
           V := arb* F;
           X(U) := V;
       end while;
```

Here, as in the case in which we want to construct a subset, the order of selection of the domain elements of X is explicit in our scheme, leading to a redundancy already noted.

To see how such specification patterns are to be used in more specific contexts, consider the case where new data types are to be defined in terms of already existing data types (broader applications to the solution of larger problems are exhibited in the following sections). For example,

(4) Total maps:

```
      find X : map(elmt E) elmt F  st domain X = E and ?
```

Using the scheme (3) to realize general map construction, we would obtain

```
      X := {};
      (while domain X /= E or not ?)
          U := arb* (E - domain X);
          V := arb* F;
          X(U) := V;
      end while;
```

which can then function as a standard scheme to realize everywhere defined maps in a most general manner. Further specifications of the form

```
      find X : totalmap(elmt E) elmt F  st P(X)
```

can then be implemented using the preceding scheme with P(X) replacing ?.

(5) One-one (and total) maps:

```
      assume E : set; F : set;
      find X : totalmap(elmt E) elmt F  st
          (forall [A, C] in X, [B, D] in X : A = B  or  C /= D) and ?
```

Using the last scheme, we obtain

```
      X := {};
      (while domain X /= E or
              (exists [A, C] in X, [B, D] in X st A /= B and C = D)
              or not ?)
          U := arb* (E - domain X);
          V := arb* F;
          X(U) := V;
      end while;
```

This version, however, requires several transformations to reach a more acceptable standard form. The main transformation is to formally differentiate the predicate '(exists ...)' appearing in the while header, which will be denoted as G(X). This predicate has a special property of 'monotonicity' with respect to the changes of X within the loop, i.e. whenever G(X) is true for some value of X, it will remain true for all subsequent values of X. If the program is to terminate, then G(X) should be kept false at all times, and we had better fail as soon as G becomes true.

To achieve this effect we move the negation of Q(X) to the end of the loop (checking that it is initially false), and making it into an assertion. This would yield

```
       X := {};
       (while domain X /= E or not ?)
           U := arb* (E - domain X);
           V := arb* F;
           X(U) := V;
           assert (forall [A, C] in X, [B, D] in X : A = B or C /= D);
       end while;
```

Next we formally differentiate the new predicate, using the property that differentiating an expression of the form

```
       assert (forall W : P);
```

amounts to asserting the differentiated quantifier. This yields

```
       X := {};
       (while domain X /= E or not ?)
           U := arb* (E - domain X);
           V := arb* F;
           assert (forall [A, C] in X : A = U or C /= V);
           X(U) := V;
       end while;
```

Next we can manipulate the resulting predicate, first by noting that A = U is false, and then by moving V 'out' of the forall condition to obtain the predicate

```
       assert  V notin {C : [A, C] in X};
```

i.e.

```
       assert  V notin range X;
```

This will yield a reasonable 'base-form' construction for one-one maps, as follows:

```
       X := {};
       (while domain X /= E or not ?)
           U := arb* (E - domain X);
           V := arb* (F - range X);
           X(U) := V;
       end while;
```

which could then be entered as the default scheme for the new data type 'oneonemap'.

(5) Enumeration: Suppose that we next want to define an enumeration of a set as follows:

```
       find X : oneonemap(elmt E) elmt {1 ... # E}  st ?
```

Using the last scheme we obtain

```
        X := {};
        (while domain X /= E or not ?)
             U := arb* (E - domain X);
             V := arb* ({1 ... # E} - range X);
             X(U) := V;
        end while;
```

We can improve this version in several ways. The one that seems to be the most natural is as follows: The original map construction scheme (3) contains a redundancy in the sense that both domain and range elements of X are selected nondeterministically. To do better, we could note that the values selected for V are always distinct from one another, so that we can simply iterate over the range of V (noting that its size is the same as the number of iterations of the original loop), and only select the next element of E nondeterministically.

Stepwise derivation of this improvement might be as follows: Formally differentiate {1 ... # E} - range X (from which V is selected), and call it MORE; also interchange the selections of U and V to obtain

```
        X := {};
        MORE := {1 ... # E};
        (while domain X /= E or not ?)
             V := arb* MORE;
             U := arb* (E - domain X);
             X(U) := V;
             MORE less:= V;
        end while;
```

Since we know (from the original scheme (3)) that one of the selections of U and V can be made deterministic (but arbitrary), we convert the selection of V into a deterministic 'arb', to obtain a reasonable base-form construction scheme for an enumeration.

It is also possible to choose the arbitrary way in which V is to be selected, to be the natural linear order of integers. This choice might not be generally suitable for an arbitrary construction of an enumeration, but in most cases will yield the desired way in which X should grow. To this end, we convert the statement 'V := arb MORE' into 'V := min/ MORE'. Moreover, we note that one always has the equality V = # X + 1. Hence we can eliminate the use of MORE altogether, and come up with the following version:

```
        X := {};
        (while domain X /= E or not ?)
             U := arb* E st U notin domain X;
             X(U) := # X + 1;       $ (evaluated right-to-left)
        end while;
```

3. AN EXAMPLE: TESTING A GRAPH FOR EXISTENCE OF CYCLES

In this section we describe a possible construction via program transformation of a program which tests a given directed graph for existence of cycles, from the high level specification suggested by Dewar et al in [De].

VERSION 1: Let G be a given directed graph. That is, G is a set of ordered pairs (edges). We are to test whether G contains a cycle. In other words, we want to check whether there exists a (nonempty) subset S of G having the property

```
(forall X in S : (exists Y in S st X(2) = Y(1) ) )
```

This can essentially be put as the following specification:

```
find S : subset G st S /= {} and
     (forall X in S : (exists Y in S st X(2) = Y(1) ) )
```

As is well known, this condition can be tested by an algorithm linear in the number of edges of the graph, to wit topological sorting (cf. [Kn, p. 258]). If directly executed the specification just written is exponential in the number of edges of G. However, we shall see that by transforming this specification we can come after several stages to a related algorithm that is linear in the number of edges of G, but nevertheless quite different from the topological sort. Some of the transformation steps used in our derivation are examined from a more formal point of view in [Sh].

The first thing that we might attempt is to construct S incrementally, following the approach described in section 2. Using the basic subset construction scheme given there, we obtain

VERSION 2:

```
S := {};
(while (exists X in S st (forall Y in S : X(2) /= Y(1) ) )
  or S = {} )
     Z := arb* (G - S);
     S with:= Z;
end while;
```

This form is amenable to formal differentiation. Indeed, let

```
P(X, S)  =  forall Y in S : X(2) /= Y(1)

BADEDGES  =  { X in S st P(X, S) }
```

Then note that adding Z to S cannot add new elements to BADEDGES (except that Z itself may belong to BADEDGES), because if W belonged to S before adding Z to it and did not satisfy P then, it still would not satisfy P. On the other hand, the insertion of Z into S could eliminate certain edges from BADEDGES, as P(X) becomes now more restrictive than before. Thus, one obtains the following version

VERSION 3:

```
      S := {};
      BADEDGES := {};
      (while exists X in BADEDGES or S = {})
          Z := arb* (G - S);
          S with:= Z;
          if (forall Y in S : Z(2) /= Y(1) ) then
              BADEDGES with:= Z;
          end if;
          (forall W in BADEDGES)
              if W(2) = Z(1) then
                  BADEDGES less:= W;
              end if;
          end forall;
      end while;
```

At this point, we would like to restrict the nondeterministic selection of Z in a way which would make it 'productive', in the sense that adding Z to S will make BADEDGES as small as possible. We note that this set can be increased by Z at most, so that we would like to balance this change by removing at least one element from it. This can be done by selecting Z such that X(2) = Z(1), for then X is certain to be removed from BADEDGES.

OBSERVATION: When applying a transformation whose effect is to limit the search space in a manner arrived at arbitrarily, one must show that the resulting program is equivalent to the previous version, in the sense that if the new version will fail, the old version will also have failed. This issue is noted here, but is not elaborated below (see however [Sh] where this subject is further discussed and where this transformation is formally justified).

This will produce the following

VERSION 4:

```
      S := {};
      BADEDGES := {};
      (while exists X in BADEDGES or S = {})
          Z := arb* {W in G - S st
              (if X /= OM then W(1) = X(2) else true end)};
          S with:= Z;
          if (forall Y in S : Z(2) /= Y(1) ) then
              BADEDGES with:= Z;
          end if;
          (forall W in BADEDGES)
```

```
            if W(2) = Z(1) then
                BADEDGES less:= W;
            end if;
        end forall;
    end while;
```

Next, a (nontrivial) verification step will prove the following facts:

(a) The cardinality of BADEDGES is 0 the first time the loop is entered.

(b) In the first iteration of the loop, BADEDGES increases by one element, or remains the same (i.e. empty).

(c) In any other iteration, BADEDGES either increases by one element or does not increase, and at the same time decreases by at least one element.

(d) As a corollary, the cardinality of BADEDGES is at all times at most one.

These facts allow us to eliminate the loop (forall W in BADEDGES) and replace it by the deletion of X from this set. Moreover, we can replace references to BADEDGES by references to its singleton element. All this will yield the following version:

VERSION 5:

```
    S := {};
    BADEDGE := OM;
    (while BADEDGE /= OM or S = {})
        Z := arb* {W in G - S st
            (if BADEDGE /= OM then W(1) = BADEDGE(2) else true end)};
        S with:= Z;
        BADEDGE := OM;     $ removing it from the set
        if (forall Y in S : Z(2) /= Y(1) ) then
            BADEDGE := Z;
        end if;
    end while;
```

Next we simplify the remaining 'forall' condition by transforming it into

```
    Z(2) notin { Y(1) : Y in S }
```

and simplify further by formally differentiating the new set expression with respect to S. This gives us the following

VERSION 6:

```
        S := {};
        BADEDGE := OM;
        PNODES := {};
        (while BADEDGE /= OM or S = {})
            Z := arb* {W in G - S st
                (if BADEDGE /= OM then W(1) = BADEDGE(2) else true end)};
            S with:= Z;
            PNODES with:= Z(1);
            BADEDGE := OM;
            if Z(2) notin PNODES then
                BADEDGE := Z;
            end if;
        end while;
```

The next simplification step is to prove that the condition 'Z notin S' in the selection of Z is redundant, and is implied by the other condition Z(1) = BADEDGE(2). This can be done by noting that if Z(1) = BADEDGE(2) then Z(1) notin PNODES (by the way BADEDGE has been computed); i.e. Z(1) notin {Y(1) : Y in S}, so that Z notin S. Thus S is not used at all during the loop, (except for the test S = {} which is relevant only for the first iteration through the loop. and can be replaced by another test), and in fact is not used at all (it is sufficient to know whether such an S exists). Hence we can eliminate S from the program altogether, getting the better version

VERSION 7:

```
        BADEDGE := OM;
        PNODES := {};
        (until BADEDGE = OM)
            Z := arb* {W in G st
                (if BADEDGE /= OM then W(1) = BADEDGE(2) else true end)};
            PNODES with:= Z(1);
            BADEDGE := OM;
            if Z(2) notin PNODES then
                BADEDGE := Z;
            end if;
        end until;
```

Next, we 'unroll' the loop so as to separate its first iteration from the others. Also, since the edges themselves are not maintained in Version 7, but only their end points, we can substitute [U, V] for Z and [UE, VE] for BADEDGE. (Note that we adopt the convention that assigning OM to a pair (such as [UE, VE]) means assigning OM to each component.) This gives

VERSION 8:

```
      [UE, VE] := OM;
      PNODES := {};
      [U, V] := arb* G;
      PNODES with:= U;
      if V notin PNODES then
          [UE, VE] := [U, V];
      end if;
      (while [UE, VE] /= OM)
          [U, V] := arb* {[U1, V1] in G st U1 = VE};
          PNODES with:= U;
          [UE, VE] := OM;
          if V notin PNODES then
              [UE, VE] := [U, V];
          end if;
      end while;
```

But if we change the test in the while loop to 'VE /= OM' then we see that UE is not used at all in the program; furthermore, U then is not used in the while loop. Thus selection of [U, V] in the loop can be reduced to a selection of V. Also the first selection of [U, V] can be broken into a selection of U (from domain G) followed by a selection of V (from G{U}). After some additional simplifications we get

VERSION 9:

```
      VE := OM;
      PNODES := {};
      U := arb* domain G;
      V := arb* G{U};
      PNODES with:= U;
      if V notin PNODES then
          VE := V;
      end if;
      (while VE /= OM)
          V := arb* G{VE};
          PNODES with:= VE;
          VE := OM;
          if V notin PNODES then
              VE := V;
          end if;
      end while;
```

But then if we rename U as VE (moving the first assignment to VE down) we note that the code from the first selection of V is identical to the code within the loop. We can then 'roll' it back into the loop, and can also eliminate the assignment of OM to VE and the test of VE in the loop header by noting that the loop will terminate iff the condition in the if statement within the loop is false. All this will produce

VERSION II:

```
      PNODES :={};
      VE := arb* domain G;
      (loop do)
           V := arb* G{VE};
           PNODES with:= VE;
           if V notin PNODES then
                VE := V;
           else
                stop;
           end if;
      end loop;
```

This version constructs all cycles in the graph in a nondeterministic manner, simply by building up a path and checking whether its last edge ends at a node already along that path. Though this is a nondeterministic, and hence inherently inefficient, program, we may nevertheless say that among all programs of this class it is the most efficient since it can succeed in a number of steps equal to the length of the smallest cycle in G and since it uses very simple data structures. However, it remains disastrously inefficient if implemented deterministically using backtracking. The next step is therefore to find a way of avoiding the nondeterministic choice of Z.

To this end, let us assume that the backtracking implied by nondeterministic selection is made explicit in our program, e.g. by holding backtracked quantities on a stack. Then a very interesting and general transformation becomes applicable. A shortcoming of naive backtracking is that it 'does not learn from mistakes', namely - it can not exploit information due to the fact that it has previously failed along some path. More sophisticated backtracking will use certain 'memo functions', to record whatever useful information is available from a failure (cf. [Sc2]). This is quite analogous to the use of memo functions to optimize recursive procedures (cf. [Co] for example). The memo functions used in backtracking should of course be nonbacktracked (corresponding to the fact that memo functions are global in the case of recursion).

In our example, a natural memo variable could be some set of nodes already visited. A sufficiently powerful verification step might then prove the following

CLAIM: If the backtracking mechanism has failed to find a cycle from a node VE, then it could not be able to find a cycle by going through VE along a later backtracking path.

In other words, once having failed to find a loop while examining a node VE, we can add it to our 'memo' set and exclude it from any further path tracing. Let BADNODES denote the memo set just described. Then the preceding computations bring us to the following version, which uses BADNODES to limit the search (see [DSc] for the explanation of the

backtracking primitives OK and FAIL used in this version):
VERSION 11:

```
        BADNODES := {};          $ the nonbacktracked memo set
        PNODES := {};
        if exists VE in domain G st VE notin BADNODES and OK then
            (loop do)
                if exists V in G{VE} st V notin BADNODES and OK then
                    PNODES with:= VE;
                    if V notin PNODES then
                        VE := V;
                    else
                        stop;
                    end if;
                else        $ failure
                    BADNODES with:= VE;
                    FAIL;
                end if;
            end loop;
        else
            print(' total failure ');
            stop;
        end if;
```

This version is already linear in the number of edges and nodes in the graph being analyzed. Nevertheless, we will still want to apply additional backtracking optimizations to it, in order to minimize the effort of stacking and unstacking environments.

Our objective is to stack as little as possible, and when we fail, to undo the changes of the values of other backtracked but unstacked variables in terms of the stacked variables.

OBSERVATION: Backtracking optimizations of this nature are apt to play a central role in the final phases of our transformational process. It seems plausible that automatic methods could eliminate most of the effort involved in these transformations, thereby making them relatively painless for our system user. (See also next section where a similar optimization is used in a transformational solution of the eight queens problem.) For a discussion of such possible optimizations, see [Sc2].

In Version 11, we can note that the only variable changed between the first backtracking point and the second one is VE, so that only VE need be saved. Between any two consecutive arrivals at the second backtracking point, the variables being changed are V, VE and PNODES. VE is used and then redefined, so that we should save it. However, V need not be saved, as it is equal to the current value of VE; also PNODES need not be saved, as it is modified only incrementally, by adding VE to it (this operation is reversible, since at the point of insertion VE does not belong to PNODES), and VE will be stacked. Finally there is no need to stack the (address of the) backtracking point, as we can determine to which backtracking point to branch after a failure by checking whether PNODES is {} (in which case we return to the first point) or not (and then return to the second one).

These considerations lead us to the following version (in which set iterations are expanded using two primitives 'zeroelmt' to initialize such an iteration, and 'nextelmt' to proceed from a given element to the next one; this is done to allow us to backtrack into a point within the (unexpanded) iteration operation):

VERSION 12:

```
       BADNODES := {};
       PNODES := {};
       STACK := [];
       DOMG := domain G;
       VE := zeroelmt(DOMG);
back1: (doing VE := nextelmt(VE, DOMG); while VE /= OM)
            if VE notin BADNODES then
                goto succeed1;
            end if;
       end doing;
       print('total failure');
       stop;
succeed1:
       STACK with:= VE;
       (loop do)
           GVE := G{VE};
           V := zeroelmt(GVE);
back2:     (doing V := nextelmt(V, GVE); while V /= OM)
                if V notin BADNODES then
                    goto succeed2;
                end if;
           end doing;
$ a failure
           BADNODES with := VE;
           V := VE;
           VE frome STACK;
           PNODES less:= VE;
           GVE := G{VE};
           if PNODES = {} then
               goto back1;
           else
               goto back2;
           end if;

succeed2:
           STACK with:= VE;
           PNODES with:= VE;
           if V notin PNODES then
               VE := V;
           else
               stop;
           end if;
```

```
    end loop;
```

We take this as our final destination along this transformational path. It is interesting to note that this version is essentially an expanded (and optimized) version of a depth-first search of the given graph. It is noteworthy that the above sequence of transformations have been chosen so as to avoid as much as possible use of 'ingenious' steps in which the next version is obtained by applying a rather deep and nonobvious transformation to the current version. If such steps were included in our process, we could obtain the topological sort from our original specification using e.g. the approach outlined in [DS].

It is nevertheless of interest to see how the topological sort could be derived by a similar sequence of transformations, starting with another specification provably equivalent to our original specification. That is, we prefer to shift the application of 'clever' transformations to the specification level, so that a considerable effort in proving the correctness of the topological sort algorithm can be elided. We now turn our attention to this latter problem.

Consider the following specification:

```
assume N = domain G + range G;
find Y : enumeration(N)  st
        (forall [A, B] in G : Y(A) < Y(B)));
```

where by enumeration we mean a one-one map from N onto {1 ... # N}. We can then use the default enumeration constructing scheme described in section 2, to obtain a first executable version:

VERSION 1:

```
Y := {};
(while domain Y /= N or exists [A, B] in G st Y(A) >= Y(B))
    X := arb* (N - domain Y);
    Y(X) := # Y + 1;      $ (evaluated right-to-left)
end while;
```

We next note that the predicate 'exists ... ' is monotone, i.e. if it ever becomes true for some value of Y, it will remain true when new elements are added to Y. Hence, to avoid immediate failure it must be kept false at all times, and consequently move it (negated) to the end of the loop. Then we formally differentiate it as follows:

```
(forall A in N : [A, X] in G implies Y(A) < Y(X) ) and
(forall B in N : [X, B] in G implies Y(X) < Y(B) )
```

A very important principle can now be exemplified. In this case we have a predicate Q(Y) whose full meaning will show only when Y is fully generated. To be able to test Q(Y) also for partial values of Y we interpret it by ignoring any subpredicate involving still undefined components of Y (i.e. interpreting such subpredicates as being OM).

However, when we formally differentiate such a predicate, we might want to 'look ahead' and consider also the relation of the newly added component of Y to the components of Y still to be added. In several cases (including our example) it may be possible to simplify the derived predicate by using general properties that characterize the still missing elements of Y.

In the case before us, we can split the nodes in N into two classes, those in domain Y, and those (including X) still outside that domain. Let Y' denote the current value of Y. This will give us the predicate

```
        (forall A in domain Y' : [A, X] in G implies Y(A) < #Y' + 1)
        and
        (forall A in N - domain Y' : [A, X] in G implies Y(A) < #Y' + 1)
        and
        (forall B in domain Y' : [X, B] in G implies #Y' + 1 < Y(B))
        and
        (forall B in N - domain Y' : [X, B] in G implies #Y' + 1 < Y(B))
```

The first conjunct simplifies to 'true', since for such A we have always Y(A) < #Y' + 1. The second conjunct can be simplified, by noting that Y(A) < #Y' + 1 is always false, so that we must have [A, X] notin G. To simplify the third conjunct, we note that #Y' + 1 < Y(B) is false there, so that we have [X, B] notin G. In the fourth conjunct, #Y' + 1 < Y(B) is true for all B except X, so that it simplifies to [X, X] notin G, which is subsumed by the simplified second conjunct. The derived predicate thus simplifies to

```
        (forall A in N - domain Y' : [A, X] notin G) and
        (forall B in domain Y' : [X, B] notin G)
```

Hence, we were able to deduce a property that X must satisfy in relation to nodes not yet selected (the nodes A above), even though their Y value is still undefined at this point. All this gives

VERSION 2:

```
        Y := {};
        (while domain Y /= N)
            X := arb* {W in N - domain Y st
                (forall A in N - domain Y : [A, W] notin G) and
                (forall B in domain Y : [W, B] notin G)};
            Y(X) := # Y + 1;
        end while;
```

We can continue to simplify as follows: Another general rule of thumb is to try to isolate the object currently being selected in the predicate that governs this selection. In our case we wish to isolate W. To do this, we try to transform the 'forall A' conjunct into

```
       W notin (+ / {G{A} : A in N - domain Y})
```

but a further look at the resulting predicate will show that it gains us nothing, because the set expression appearing there is not amenable to formal differentiation. A better way would be to transform it into

```
       (forall A in N - domain Y : A notin G-1{W} )
```

and then to

```
       G-1{W} subset domain Y
```

which is in a much better shape for formal differentiation. The second 'forall' conjunct presents no problems, and we can transform it into

```
       W notin (+ / {G-1{B} : B in domain Y})
```

We have thus obtained

VERSION 3:

```
       Y := {};
       (while domain Y /= N)
           X := arb* {W in N - domain Y st
               W notin (+/ {G-1{B} : B in domain Y}) and
               W in {A in N : G-1{A} subset domain Y}};
           Y(X) := # Y + 1;
       end while;
```

Next we apply formal differentiation to both set expressions appearing in the predicate above. Call the first set PREVS and the second set NOPREDS. PREVS is easy to differentiate; NOPREDS is somewhat trickier: When increasing the domain of Y by X, no elements need be deleted from NOPREDS, and the only elements that can be added to it are those A for which A in G{X}. This observation yields the following

VERSION 4:

```
       Y := {};
       PREVS := {};
       NOPREDS := {A in N : G-1{A} = {}};
       (while domain Y /= N)
           X := arb* {W in N - domain Y st
               W notin PREVS and W in NOPREDS};
           Y(X) := #Y + 1;
           PREVS +:= G-1{X};
           NOPREDS +:= {A in G{X} : G-1{A} subset domain Y};
       end while;
```

Next we can prove that at the point of selection of X, PREVS subset domain Y, so that the test 'W notin PREVS' is redundant, and can therefore be eliminated. This will make PREVS dead, so that we can eliminate it altogether.

Then we define a new set NEWNOPREDS as NOPREDS - domain Y (this is the set from which X is selected). The next step is to formally differentiate NEWNOPREDS. To do this, we have to change the condition

```
      G-1{A} subset domain Y
into
      # {W in G-1{A} : W notin domain Y} = 0
```

Let us define, for each A in N, this expression as NUMPREDS(A). This calls for the formal differentiation of NUMPREDS(A) for all A in N. All this will produce the following

VERSION 5:

```
      Y := {};
      NUMPREDS := {};
      (forall A in N)
          NUMPREDS(A) := # G-1{A};
      end forall;
      NEWNOPREDS := {A in N : NUMPREDS(A) = 0};
      (while domain Y /= N)
          X := arb* NEWNOPREDS;
          (forall A in G{X})
              NUMPREDS(A) -:= 1;
          end forall;
          NEWNOPREDS +:= {A in G{X} : NUMPREDS(A) = 0};
          NEWNOPREDS less:= X;
      end while;
```

Next we want to remove the nondeterminism in the selection of X. This can be done by proving that any selection will succeed iff the graph does not contain a cycle. Then we can select X in a deterministic manner giving what is essentially Knuth's topological sort algorithm. It is interesting to note that our termination test for the while loop is different from the one used in the standard topological sort (i.e. 'while NEWNOPREDS /= {}'). Our test causes the program to fail (if there are cycles) during selection from an empty NEWNOPREDS; the standard test avoids such a failure, but an additional test at the loop exit is then needed, to test whether domain Y /= N. Hence the difference between these two versions is that these two tests are executed in a different order.

A gratifying by-product of our transformational process is that it implies easily that our final program can compute all possible topological sorted orderings of N. This is because the original specification had that property, and the search space has not been pruned along our transformational process (except for the conversion of nondeterministic selection to a deterministic but arbitrary one).

4. ANOTHER EXAMPLE: THE EIGHT QUEENS PROBLEM

In this section we describe a possible transformational construction of a program solving the eight queens problem from its obvious 'specification', resulting in a variant of Wirth's algorithm, as appears in [Wi]. As in the previous section, the transformations have been chosen in a manner which we hope will be amenable to a large degree of mechanization (or at least formalization). General comments projecting from the experience with this problem toward the future design of a transformational system are noted as 'observations' along the way.

VERSION 1: (informal specification) Place 8 queens on an 8x8 board such that no two queens can attack each other.

In this informal initial version several concepts require more formal definitions, such as '8x8 board' and 'two queens can attack each other'. There is also one fundamental design issue: What does 'place 8 queens' mean? It might mean: find a map from a set of 8 abstract objects ('queens') to the set of board positions. However, using the fact that these queens are indistinguishable from one another, and the fact that no two queens can occupy the same board position, we deduce that the above request might also mean: find a set of 8 board positions. The second request is more general in the sense that it does not impose order among the queens, but is also more specific in the sense that it limits the search space. We will consider the second form of the problem, as this will make it easier for us to convert nondeterministic selection to a deterministic selection later in the transformational process.

VERSION 2: (formal specification): Let B denote the set of all board positions, i.e. B = N x N, where N = {1 ... 8}. Let ATT denote a relation on B such that forall X, Y in B : ATT(X, Y) means that a queen at position X can attack position Y. I.e.

```
      ATT(X, Y) =  X - Y in ATTACK
where
      ATTACK = ATTACKC + ATTACKR + ATTACKUD + ATTACKDD
where
      ATTACKR = {[m - n, 0] : m in N, n in N}    (attack in row)
      ATTACKC = {[0, m - n] : m in N, n in N}    (in column)
      ATTACKUD = {[m - n, m - n] : m in N, n in N}  (in up diagonal)
      ATTACKDD = {[m - n, n - m] : m in N, n in N}  (in down diagonal)
```

Then

```
      find S : subset B st
           # S = 8  and
           (forall Y in S, Z in S : Y /= Z implies not ATT(Y, Z))
```

OBSERVATION: We assume that our system has already been given some 'education' of a general kind which is relevant to some of the formal structural features of the 8 queens problem, i.e. that it can recognize and deal effectively with cartesian products of integers, also that it can recognize X - Y in the definition of ATT as a vector subtraction, and also that it knows the basic rules concerning such operations. A main aim in studying examples such as the 8 queens problem is to form some feeling for the classes of very high level program manipulations likely to be of broad utility.

Our first transformation uses the basic subset construction scheme given in section 2 to convert the above version into the following executable form:

VERSION 3:

```
      S := {};
      (while # S < 8 or
              exists Y in S, Z in S st Y /= Z and ATT(Y, Z))
         X := arb* (B - S);
         S with:= X;
      end while;
```

We next note that the predicate 'exists ...' is monotone (in the sense described in section 3), so that it must be kept false at all times. This implies that if we formally differentiate this predicate, then its derivative must also be false, and this fact can be used as a constraint in the selection of X. This will yield the following version:

VERSION 4:

Let B, N, ATT etc. be defined as above.

```
      S := {};
      (while # S < 8)
         X := arb* {W in B - S st
             W /= W implies not ATT(W, W) and
             (forall Y in S :
                W /= Y implies not ATT(W, Y) and
                Y /= W implies not ATT(Y, W) )};
         S with:= X;
      end while;
```

This can however be simplified, by noting that W /= W is false, and that W /= Y and Y /= W will always be true. also we utilize the symmetry of the relation ATT, to eliminate one appearance of ATT in Version 4. We thus obtain

VERSION 5:

```
      Same definitions as above

      S := {};
      (while # S < 8)
          X := arb* {W in B - S st (forall Y in S : not ATT(W, Y))};
          S with:= X;
      end while;
```

Next we substitute (another transformation) the definition of ATT to transform

```
      not ATT(W, Y)
into
      W - Y notin ATTACK
```

and by a further substitution into

```
      W - Y notin (ATTACKR + ATTACKC + ATTACKUD + ATTACKDD)
```

OBSERVATION: It is useful to break a definition (such as that of ATT) into several subdefinitions, so that they can be applied separaely, allowing control over the degree of expansion ('unfolding' if you will) during substitution.

Next apply a set-theoretic rule of the form

```
       Z notin (A + B)  =  Z notin A and Z notin B
```

and associativity of set union to obtain

VERSION 6:

```
      S := {};
      (while # S < 8)
          X := arb* {W in B - S st
              (forall y in S :
                  W-Y notin ATTACKR and
                  W-Y notin ATTACKC and
                  W-Y notin ATTACKUD and
                  W-Y notin ATTACKDD )};
          S with:= X;
      end while;
```

Next substitute B = N x N. An interesting transformation then becomes applicable, namely - nondeterministic selection of an element of a cartesian product A x B is equivalent to a nondeterministic selection of the first component from A followed by a nondeterministic selection of the second component from B. In conjunction with this transformation, we also substitute [XC, XR] for X and [YC, YR] for Y. These substitutions, unlike substitutions of definitions, require verification of their enabling conditions, which state that X and Y should have 'data types' which permit such substitutions.

OBSERVATION: Our system will then have to maintain some kind of type information concerning the variables appearing in the program being constructed. This should generally be much simpler to do as compared e.g. to the type analysis currently used in the SETL optimizer (cf. [16]), especially when the initial specification is control-free.

Applying also the definition of vector subtraction we obtain

VERSION 7:

```
      S := {};
      (while # S < 8)
          XC := arb* N;
              XR := arb* {WR in N st [XC, WR] notin S and
                  (forall [YC, YR] in S :
                      [XC-YC, WR-YR] notin ATTACKR and
                      [XC-YC, WR-YR] notin ATTACKC and
                      [XC-YC, WR-YR] notin ATTACKUD and
                      [XC-YC, WR-YR] notin ATTACKDD )};
          S with:= [XC, XR];
      end while;
```

Next, substitute the definition of ATTACKR, ATTACKC etc. Consider ATTACKR for example. We get

```
      [XC-YC, WR-YR] notin {[m - n, 0] : m in N, n in N}
```

which, after application of a few rules will get transformed into

```
      WR /= YR
```

Similar inequalities can be obtained from the other subrelations. Hence we obtain

VERSION 8:

```
      S := {};
      (while # S < 8)
          XC := arb* N;
              XR := arb* {WR in N st [XC, WR] notin S and
```

```
                (forall [YC, YR] in S :
                    WR /= YR and XC /= YC and
                    XC-YC /= WR-YR and XC-YC /= -(WR-YR) )};
        S with:= [XC, XR];
    end while;
```

NOTE: At this point, B, ATTACKR, ATTACKC etc. become dead and can therefore be eliminated.

Next use the rule

```
    (forall Z in S : P(Z) and Q(Z))  =
    (forall Z in S : P(Z)) and (forall Z in S : Q(Z))
```

and the rule

```
    (forall Z in S : A /= F(Z))  =  A notin {F(Z) : Z in S}
```

and the rule

```
    A - B /= C - D   =   A - C /= B - D
```

and similar rules concerning addition, to change the predicate appearing in the last version to

```
    WR notin {YR : [YC, YR] in S} and
    XC notin {YC : [YC, YR] in S} and
    WR-XC notin {YR-YC : [YC, YR] in S} and
    WR+XC notin {YR+YC : [YC, YR] in S}
```

At this moment we can get rid of the test [XC, WR] notin S by proving that this is implied by either of the predicates

```
    WR notin {YR : [YC, YR] in S}
```

or the second one.

Now we are in a position to apply formal differentiation to the sets appearing above. Calling these sets BADR, BADC, BADUD and BADDD respectively, we obtain the following

VERSION 9:

```
    S := {};
    BADC := BADR := BADUD := BADDD := {};
    (while # S < 8)
        XC := arb* N;
            XR := arb* {WR in N st
                XC notin BADC and WR notin BADR and
                WR-XC notin BADUD and WR+XC notin BADDD};
        S with:= [XC, XR];
        BADC with := XC;
```

```
          BADR with := XR;
          BADLD with := XC-XR;
          BADDD with := XC+XR;
      end while;
```

Next perform code motion, moving the code independent of XR to the point before the selection of XR. Also, formally differentiate # S appearing in the while clause, to obtain a fragment which has the form

```
      NS := 0;
      (while NS < 8)
          NS +:= 1;
          XC := arb* {WC in N st WC notin BADC};
          BADC with:= XC;
          block(XC);
      end while;
```

Then a very interesting transformation becomes applicable. This transformation eliminates the nondeterminism in the choice of XC. In general, if one has the pattern

```
      K := {};
      (forall ITERATOR)
          X := arb* (A - K);
          K with:= X;
          BLOCK(X);
      end;
```

where the ITERATOR's variable(s) do not appear in the loop except to modify themselves, and where, for any two values X1, X2 of X chosen in succession, we have the property that the effect of executing BLOCK(X1) followed by BLOCK(X2) is the same as the effect of executing BLOCK(X2) followed by BLOCK(X1), and if the number of times the loop is executed is equal to # A then the above pattern can be transformed into

```
      (forall X in A)
          BLOCK(X);
      end;
```

That is, the X's chosen are all the elements of A, each chosen exactly once, and the order in which they are selected is not important. Admittedly, this is the toughest transformation applied so far in our chain, and is one which requires a lot of verification concerning its enabling conditions. We would like very much to see a cleaner way of eliminating this nondeterminism.

We thus come to the following

VERSION 10:

```
      S := {};
      BADR := BADUD := BADDD := {};
      (forall XC in N)
          XR := arb* {WR in N st
              WR notin BADR and WR-XC notin BADUD and
              WR+XC notin BADDD};
          S with:= [XR, XC];
          BADC with:= XC;
          BADUD with:= XR-XC;
          BADDD with:= XR+XC;
      end forall;
```

We are now almost at our final version. The last major transformation still to be tackled involves backtracking optimization of the sort mentioned in the previous section. More precisely, we would like to make the backtracking (implied by the nondeterministic selection of XR) explicit, and optimize the environment-saving mechanism by saving as few objects as possible, and maintaining other objects (which also may have to be saved by default) in terms of the saved objects. To this end we can proceed as follows:

Since XR is the variable chosen nondeterministically, it (or, rather, a pointer to its position in N) will have to be saved. We then note that when backtracking to a previously saved environment, the only changes that took place since that save are to XC, S, BADR, BADUD, BADDD, and all of these changes are incremental and can be reversed also in an incremental fashion. (This is true if one assumes that the linear order of iteration through N will be used. Also, the inverse operation of, say, 'BADR with:= XR' is 'BADR less:= XR' only because at the point of insertion XR did not belong to BADR (which can be verified).) All this will produce

VERSION 11:

```
      S := {};
      STACK := [];
      BADC := BADUD := BADDD := {};
      XC := 0;
      (doing XC +:= 1; while XC <= 8)
          XR := 0;
back:     XR +:= 1;
          if XR > 8 then     $ failure
              if STACK = [] then     $ total failure
                  print('no solution');
                  stop;
              end if;
              XR frome STACK;
              XC -:= 1;
              S less:= [XC, XR];
              BADR less:= XR;
```

```
           BADUD less:= XR-XC;
           BADDD less:= XR+XC;
           goto back;
        elseif XR in BADR or XR-XC in BADUD or XR+XC in BADDD then
           goto back;
        else
           STACK with:= XR;
           S with:= [XC, XR];
           BADR with:= XR;
           BADUD with:= XR-XC;
           BADDD with:= XR+XC;
        end if;
     end;
```

The next thing that we can do is to note that S is not used at all in the loop (except for modifying itself). We can prove that at exit from the loop one has

S = { [I, STACK(I)] : I in [1...8]}

and consequently compute S this way at exit from the loop. This final version would be quite close to Wirth's algorithm.

ACKNOWLEDGEMENT. My thanks are given to Jack Schwartz and Ed Schonberg for reviewing this paper.

REFERENCES

[Ba] Bauer, F.L. et al., "Systematics of Transformation Rules", Lecture Notes in Comp. Sci. 69, Springer, Berlin 1979.

[BD] Burstall, R.M. and Darlington, J., "A Transformation System for Developing Recursive Programs", JACM 24(1977) 44-67.

[Co] Cohen, N.H., "Characterization and Elimination of Redundancy in Recursive Programs", Proc. 6th POPL Conf. (1979) 143-157

[De] Dewar, R.B.K., Grand, A., Liu, S.C., Schonberg, E. and Schwartz, J.T., "Programming by Refinement, as Exemplified by the SETL Representation Sublanguage", TOPLAS 1(1979).

[DS] Dewar, R.B.K. and Schonberg, E., "The Elements of SETL Style", Proc. ACM Conf. Detroit (1979)

[DSc] Dewar, R.B.K. and Schwartz, J.T., "Syntax and Semantics of a Restricted Bactrack Implementation", SETL Newsletter 186, Courant Institute 1977.

[Dea] Deak, E., Ph.D. Thesis, New York University (in progress).

[Kn] Knuth, D.E., "The Art of Computer Programming", Vol. I Addison-Wesley, 1973.

[LGdR] Lee, S., Gerhart, S.L. and deRoever, W.P., "The Evolution of List-Processing Algorithms", Proc. 6th POPL conf. (1979) 53-67.

[Lo] Low, J.R., "Automatic Data Structure Selection: An Example and Overview", CACM 21(1978), 376-384.

[Pa] Paige, R., "Expression Continuity and the Formal Differentiation of Set-Theoretic Expressions", Courant Comp. Sci. Report 15, 1979.

[SSS] Schonberg, E., Schwartz, J.T. and Sharir, M., "Automatic Data Structure Selection in SETL", Proc. 6th POPL Conf. (1979) 197-210.

[Sc1] Schwartz, J.T., "Correct Program Technology", in "Les Fondements de la Programmation" Toulouse (1977) 225-269.

[Sc2] Schwartz, J.T., "Intermediate result recording and other Optimization Techniques for Recursive and Backtrack Programs", SETL Newsletter 155, Courant Institute 1975.

[Sh] Sharir, M., "Some Observations Concerning Formal

Differentiation of Set-Theoretic Expressions", Tech. Rept. Courant Institute 1979 (to appear).

[Te] Tenenbaum, A.M., "Type Determination in Languages of Very High Level", Courant Comp. Sci. Report 3, 1974.

[WS] Walker, S.A. and Strong, H.R., "Characterization of Flowchartable Recursions", J. Comp. Sys. Sci. 7(1973) 404-447.

[Wi] Wirth, N., "Program Development by Stepwise Refinement", CACM 14(1971) 221-227.

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

~~JAN 3 1 [illegible]~~			
GAYLORD 142			PRINTED IN U S A